National Institute of Standards and Technology
Technology Administration, U.S. Department of Commerce

ProtocolSessions in the GMPLS Lightwave Agile Switching Simulator (GLASS)

Version: Draft 1.0

ABSTRACT:

The GMPLS Lightwave Agile Switching Simulator (GLASS) is an extension of the Scalable Simulation Framework Network (SSFNet) and provides the framework for optical components and the GMPLS structure. GLASS also provides a set of protocols that are ready to use and that gives a new user an idea about how to use this new framework and how to implement protocols.

This document contains a collection of the protocols that can be found in the GLASS simulator.

TABLE OF CONTENT

TABLE OF TABLES

TABLE OF FIGURES

1 INTRODUCTION

GLASS provides a set of protocols that have been implemented to test and validate the framework. Protocols are called "ProtocolSession" in SSFNet. In the first part of this document you will find characteristics of the SSFNet protocol implementation. In the second part, you will find a set of protocols using the GLASS framework and how to configure them. To illustrate this description, it contains samples of DML files.

2 PROTOCOLSESSION IN SSFNET

This chapter introduces the concepts of so called "ProtocolSession" in SSFNet. More detailed information is published in the document "Protocol modeling with SSF.OS" [1].

To start, the figure below shows the graphical concept of the classes **ProtocolGraph**, **ProtocolSession**, **ProtocolMessage**, and **PacketEvent**.

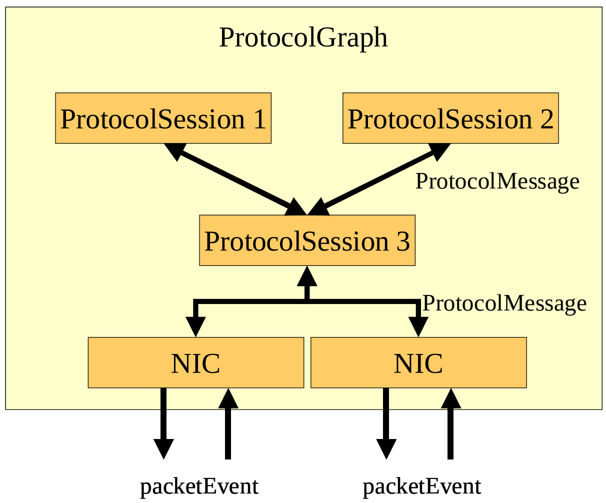


Figure 1: ProtocolGraph

The **ProtocolGraph** is the container for all the protocols available in a node. This container configures and initializes the protocols when running a simulation. There is no specific configuration of a graph. The user can configure it in the DML file.

The **ProtocolSession** is the instance of the service running in a node (for example IP). It is configurable via DML file.

The **ProtocolMessage** is the entity that is exchanged between two protocols via the method *push(ProtocolMessage msg, ProtocolSession fromSession)*.

The **PacketEvent** is the entity used to exchange information between nodes. There is an easy conversion done in the queues to map *ProtocolMessage* and *PacketEvent*.

In this document, there is no explanation on how to implement your protocol. To get this information, look in the document [2].

3 PROTOCOLSESSION IN GLASS

This chapter presents a collection of protocols that are implemented using the GLASS framework. For each subsection, there is a description of the protocol, its package, its configuration and also examples of the configuration.

3.1 THE PROTOCOL SIMPLE PROTOCOL

3.1.1 DESCRIPTION

The class **SimpleProtocol** is one of the first protocols implemented on the new framework. Its role idea was to test and validate the behavior of the optical components. This protocol is sitting on top of the *OXCSwitch* [3] and does not need IP. It requests a light path by using the utilities in the package **gov.nist.antd.optical.util** to a destination node and sends data. This protocol does not need any signaling to establish a lightpath.

3.1.2 PACKAGE AND RELATED CLASSES

The **SimpleProtocol** is located in the package **gov.nist.antd.merlin.protocol.sample**.
The class **SimpleProtocolHeader** contains the implementation of the header of this protocol. This class is located in the same package.

3.1.3 VERSION AND BUILD

This document refers to the build 20430. The minimum versions are:

- **SimpleProtocol**: v1.4
- **SimpleProtocolHeader**: v1.2

3.1.4 DML SCHEMAS

This section shows the DML schema for the *SimpleProtocol* and explains how to configure it.

ProtocolSession [
  name simpleProtocol
  use gov.nist.antd.merlin.protocol.sample.SimpleProtocol
  destination %I
  routing %S
  wavelength %S
  bandwidth %S
  delay %F
]
	Attributes name and use are the same as any ProtocolSession.
	
	Optional attribute destination specifies the destination node of the messages (default value: -1). The default value will allow the node to receive messages but not to send.
	Optional attribute routing specifies the name of the routing protocol (default value: ShortestPathDistance).
	Optional attribute wavelength specifies the name of the wavelength algorithm (default value: BestFit).
	Optional attribute bandwidth specifies the bandwidth request for the connection (default value: 2.5 Gbps).
	Optional attribute delay specifies the delay between two transmissions (default value 0.1s).

Table 1: The Configuration Schema of the SimpleProtocol

As for all protocols, the “SimpleProtocol” has to be specified in all the nodes that are going to send and receive *SimpleProtocol* messages. Nodes that are in the path of the message do not need to have this protocol installed because the path is O/O/O switched.

To use the default values, the DML file must contain at least an algorithm named ShortestPathDistance and another one named BestFit.

3.1.5 IMPLEMENTATION

Once the instance of **SimpleProtocol** is configured to send messages, its behavior is as follow:

- Creation of the quality of service that matches the configuration.
- Request of a connection by using the instant lightpath establishment. This is provided by the class **gov.nist.antd.merlin.util.ConnectionUtil**. In other words there are no signaling messages to establish a path.
- If a path is found, register to it.
- Send messages with the delay that is given as input in the configuration.

Between two transmissions, the protocol unregisters the lightpath after (delay/2). Then it requests the connection again. This is done to free unused resources. The light path can change during the simulation because the protocol is only interested in sending data, not in using the same lightpath all over again.

The content of the message sent is “Hello from <sourceID> to <destinationID>”.

3.1.6 EXAMPLE

The following DML is the configuration of a Node that is going to send messages to node 18 every 0.1sec. The algorithms used are *ShortestPathSRLG* to compute the route and *BestFit* to assign the wavelength.

```
ProtocolSession [
     name hello
     use gov.nist.antd.merlin.protocol.sample.SimpleProtocol
     routing shortestPathSRLG
     wavelength bestFit
     destination 18
     delay 0.1
]
```

Table 2: The Example for the SimpleProtocol

3.2 PROTOCOL NEIGHBOR DISCOVERY

3.2.1 DESCRIPTION

This protocol is implemented in the class **OptNeighbour**[*] and represents a simple signaling protocol. The goal for it is to create a table of neighbors for a node and how they can be reached. This protocol is working on optical links only.

3.2.2 PACKAGE AND RELATED CLASSES

The package that contains the protocol is **gov.nist.antd.merlin.protocol.discovery**.
Three classes are located inside this package:

- **OptNeighbour**: This class contains the implementation of the signaling protocol itself.
- **NeighbourTable:** It is the table used by the protocol above to store the information about all the neighbors.
- **NeighbourHeader**: This class is the message header that contains the information that has to be exchanged between the nodes.

3.2.3 VERSION AND BUILD

The current build of the package is 20430. The minimum versions are:

- **OptNeighbour**: v1.1
- **NeighbourTable**: v1.0
- **NeighbourHeader**: v1.2

[*] Some classes are spelled in British English in lieu of American English.

3.2.4 DML SCHEMAS

This following table presents the configuration schema of the protocol.

`ProtocolSession [`
`  name discovery`
`  use gov.nist.antd.merlin.protocol.discovery.OptNeighbour`
`  _extend gov.nist.antd.merlin.util.AutoConfigCtrl`
`]`
	Standard configuration
	
	Allows the configuration of the add-drop-ports.

Table 3: The Configuration Schema for the OptNeighbour

3.2.5 IMPLEMENTATION

This chapter presents the implementation and the behavior of the protocol.

The class **OptNeighbour** extends the class **AutoconfigCtrl**. This class, as described in [3], allows the user to configure the add-drop-ports and specify which lambda is going to be used to send or receive information. If the user does not enter any manual configuration, the protocol will configure the *OXCSwitch* automatically to be connected to all available input control lambdas as well as to all available output control lambdas.

There are multiple steps in the protocol. The first one is to send signaling messages to all the neighbors to notify their presence. This is done in a global broadcast. Then each node will update the information in the message and make a global broadcast to reach the sender of the message. With this procedure, all possible ways to reach the original sender of the message are tried. A node gets the information about which port to use to send a data to a specific port to a specific neighbor. When a node receives twice the same information it will not forward it to its neighbors.

The *NeighbourHeader* contains some specific fields to specify if this is a notification or a response message.

3.2.6 EXAMPLE

Two examples are available in the directory “glass/examples/optical”.

- **SimpleOpticalNeighbourDiscovery.dml**: A four-node network and each of the nodes contains the neighbor protocol. The configuration of the add-drop-port is done manually.
- **SimpleOpticalNeighbourDiscovery2.dml**: Same example as the previous one, but the configuration of the add-drop-ports is done automatically.

3.3 BACKUPMANAGER

3.3.1 DESCRIPTION

The backup manager that is implemented in the class **BackupManager**, is a simple implementation of a backup protocol. The mechanism is as follow:

> When a failure/recovery occurs, the backup manager computes a new route with a predefined algorithm. This is defined in the configuration. If another route is available, the messages will be sent, by using this new route. If no other route is available, the connection is lost.

The backup manager is a protocol that must be installed in every node where a protection is required. The manager can use algorithms that have knowledge of the whole topology. The backup manager is for link protection only. The chosen backup route is a route that is computed around the failed link and not necessarily on the entire path.

3.3.2 PACKAGE AND RELATED CLASSES

The backup protocol and the related classes or available in the package:

> **gov.nist.antd.merlin.protocol.protectionlink**.

This package contains also an example algorithm (**BackupLink**) based on the algorithm **ShortestPathDistance** and can be used to compute the backup route for the failed link. The last class (**LinkGraph**) is the graph that represents the whole topology. The algorithm that computes the backup routes uses this graph.

3.3.3 VERSION AND BUILD

This document refers to the build 20430 and later. The minimum versions are:

- **BackupManager**: v1.3
- **BackupLink**: v1.2
- **LinkGraph**: v1.1

3.3.4 DML SCHEMAS

This section describes hoe to configure the protocol and its detailed implementation.

ProtocolSession [
  name restoration
  use gov.nist.antd.merlin.protocol.protectionlink.BackupManager
  BACKUPALGORTITHM $S1!
]
	Standard protocol configuration
	
	Attribute BACKUPALGORITHM specifies the name of the algorithm to use to compute the backup of the link.

Table 4: The Configuration Schema for the BackupManager

The attribute "BACKUPALGORITHM" is a static class attribute. This means that the configuration of this attribute in the DML file has to be done only once. If specified more than once, the last configured value becomes the global value used. The backup algorithm must also be configured in the Algorithm section of the DML file.

3.3.5 IMPLEMENTATION

The class **BackupManager** extends the class **gov.nist.antd.merlin.util.AbstractCallback.** This super class implementation registers the backup manager to the optical network interface cards (ONIC) and is notified when a failure/recovery event occurs. See the implementation of this class for more details about the registration and notification messages.

When a failure occurs, the interface notifies the *BackupManager* about the type of failure on the affected link, fiber, or lambda. The manager calls the backup algorithm to compute the new route around the failed link. If a new one is available, the manager merges both routes to create a complete new route. Then, it calls the wavelength assignment used on the primary path to compute a new path. If a new path is available, then it registers the user of the working route to the backup route using the same add-drop-ports. This ensures that the recovery mechanism is totally transparent to the upper layer protocols.

If the backup route cannot be computed, then the protocol receives an error when trying to send information. This means the connection has failed.

The implementation of this backup manager for link protection is just an example that does not cover all possibilities. Furthermore, using the mechanism of merging both routes, provided in class **gov.nist.antd.merlin.util.ConnectionUtil**, may produce some exceptions.

The algorithm **BackupLink** is a modified version of the algorithm **ShortestPathDistance**. It is used to compute the backup route. The difference between both algorithms is that the **BackupLink** creates backup routes for all links in the topology. Some backup routes may not be computed because of bandwidth, failure, or unidirectionality of the links. To compute the backup, the algorithm **BackupLink** is based on the graph created by the class **LinkGraph**.

Then during the simulation, the backup manager requests the route between the two nodes connected by the link that failed). The algorithm **BackupLink** returns the backup route if one has been computed before. Any implementation of algorithms that uses the standard interface for the algorithms can be used to compute the backup route.

3.3.6 EXAMPLE

A complete example using the *BackupManager* is available in the file **BackupManager.dml**. This file is located in the examples directory “glass/examples/optical”.

This example describes a simple three-node network that uses the algorithm **ShortestPathDistance** for the calculation of the backup routes. The traffic will be generated by the protocol **SimpleProtocol**.

3.4 TopologyManipulator

3.4.1 Description

The class **TopologyManipulator** is a pseudo-protocol because it does not send or receive any data. It is located in one or more node and uses an event generator to create failures or other events according to statistical information.

3.4.2 Package and related classes

The package **gov.nist.antd.merlin.generator.event** contains the protocol, the interfaces to use for the events and also an example of an event generator.

3.4.3 Version and Build

This document refers to the build 20430 and later. The minimum versions are:

- **TopologyManipulator**: v1.2, the ProtocolSession
- **EventGenerator**: v1.1, the interface for the event generators
- **EventParameter**: v1.1, the event information created by an event generator
- **RandomEventGenerator**: v1.2, an implementation of event generator.

3.4.4 DML schemas

This paragraph explains the configuration of the protocol.

ProtocolSession [
  name topologyManipulator
  use
  gov.nist.antd.merlin.generator.event.TopologyManipulato
  r
  generator [use $S1!]
]
	DML fragment for the topology manipulator.
	
	Attributes name and use are the same as any ProtocolSession.
	
	Attribute generator defines which module will be used to generate events.

Table 5: The Configuration Schema for the TopologyManipulator

3.4.5 Implementation

The class specified in the generator section (see Table 5) must extend the interface **EventGenerator** to be used by the topology manipulator. This interface provides two methods similar to the class

ProtocolSession. The first one is called during the configuration (method *public void config (Configuration cfg)*) and the second at the beginning of the simulation (method *public void init()*).

Basically, a generator creates an instance of the class **EventParameter** and passes it to the class **TopologyManipulator**. This class checks the information and creates the correct simulation timer to execute the event at the specific time.

The class **TopologyManipulator** can process multiple types of event (see coding), however at this time the class **RandomEventGenerator** only creates node failure events.

The **EventParameter** allows a generator to specify the object to modify, the modification type, the value to apply and the time.

The implementation of **RandomEventGenerator** creates a node failure or node recovery every second. The value of the modification is random. If the random value is true then the node is failed. It is possible to create two successive events with the same value.

3.4.6 EXAMPLE

There is no example in the GLASS directory but the following configuration may be added in any protocol graph to add the **TopologyManipulator** and the **RandomEventGenerator**:

```
ProtocolSession [
  name topologyManipulator
  use gov.nist.antd.merlin.generator.event.TopologyManipulator
  generator [ use gov.nist.antd.merlin.generator.event.RandomEventGenerator]
]
```

3.5 TrafficManager

3.5.1 Description

The class **TrafficManager** is a protocol session that requests connections and sends messages. The algorithm that determines the rules on how often connections have to be created and to whom depends on the attached generator.

3.5.2 Package and related classes

The classes are located in the package **gov.nist.antd.merlin.generator.traffic**.

The following list shows all classes that are contained in this package:

- **TrafficManager**: The protocol session that requests the connection and sends the message.
- **TrafficParameter**: This class contains the information that is needed by the TrafficManager to create the connection (quality of service, destination, message).
- **TrafficGenerator**: This interface class has to be implemented by the traffic generator.
- **RandomTrafficGenerator**: A simple implementation of a traffic generator.

3.5.3 Version and Build

This document refers to the build 20430. The minimum versions are:

- **TrafficManager**: v1.3
- **TrafficParameter**: v1.1
- **TrafficGenerator**: v1.1
- **RandomTrafficGenerator**: v1.2

3.5.4 DML SCHEMAS

This paragraph explains the configuration of the protocols

<pre>ProtocolSession [
  name trafficManager
  use gov.nist.antd.merlin.generator.traffic.TrafficManager
  generator [use $S]
]</pre>
	Basic DML structure for the traffic manager
	The traffic manager will send information according to the given generator. If not précised, then it will only receive messages.

Table 6: DML Configuration Schema

The traffic generator may contain its own DML configuration. For example, the class **RandomTrafficGenerator** is activated by specifying “active true” in the DML configuration of the generator (3.5.6).

3.5.5 IMPLEMENTATION

The class specified in the *generator* section must implement the interface **TrafficGenerator** to be able to be used by the traffic manager. This interface provides two major methods similar to the ProtocolSession. The first method is called during the configuration. It is the method *public void config (Configuration cfg)*. The second method *public void init()* is called at the beginning of the simulation. According to statistical information, the traffic generator creates an instance of **TrafficParameter** and passes it to the **TrafficManager**.

The traffic manager then will create the connection request from the node where the instance is running to the destination specified in the **TrafficParameter**. Once a path is computed, the traffic manager will start sending the messages. To be able to receive a message, it is necessary that there is an implementation of the traffic manager installed in the protocol graph of the destination. In the case of the RandomTrafficGenerator implementation, the destination of the messages is chosen randomly. This means that an instance of **TrafficManager** must be installed in each Optical Cross-Connect (OXC). The current implementation creates a TrafficParameter every 10^9-simulation-tics, which may not be appropriate for all simulations. It is up to the user to modify and/or optimize the generator.

! It is important to note that the traffic manager does not free the resources used by the connection. It is up to the generator to know when a connection must be removed.

3.5.6 EXAMPLE

Currently there is no example using the traffic manager. An easy way to include it in a simulation is to add the following DML fragment at the end of the DML file:

```
global [
  ProtocolSession [
                  name trafficManager
                  use gov.nist.antd.merlin.generator.traffic.TrafficManager
                  generator [ use
                  gov.nist.antd.merlin.generator.traffic.RandomTrafficGenerator
                  active true]
            ]
]
```

Table 7: The traffic manager as a global protocol definition

This coding will automatically add the traffic manager in all the nodes of a network and af the active attribute is true, all instances will request a connection.

3.6 THE PROTOCOL DYNRECOVERY

3.6.1 DESCRIPTION

This document describes the current implementation of the protocol DynRecovery (Dynamic Recovery) in GLASS. DynRecovery is a dynamic Route Recovery Protocol for optical networks consisting of OXCEdgeRouters and of OXCs.

3.6.2 PACKAGE AND RELATED CLASSES

The protocol **DynRecovery** and related classes are located in the package **gov.nist.antd.merlin.protocol.signaling**. The header for the protocol is implemented in the class **DynRecoveryHeader**.

3.6.3 VERSION AND BUILD

The last recent version is 1.6 and the last release is version 1.4 in build 20430 and higher. This section presents the protocol in detail.

3.6.4 CURRENT IMPLEMENTATION

3.6.4.1 ANALYSIS OF THE CLASS DYNRECOVERYHEADER

The following table lists the content of the header fields.

`byte type`	Type of the message
`int routeID`	Route on which the failure occurred
`int channelID`	Channel on which the failure occurred
`int segmentIndex`	Segment on which the failure occurred
`int currentSegInd`	Segment on which this DynRecovery was received.
`int oxcDoneID`	Value -1 if not used; valid only in OXC_DONE messages.

Table 8: The Context of DynRecoveryHeader

The corresponding size is 21 bytes. This can be requested by the method bytecount(). Assuming that an “int” uses 32 bits.

The type can be found in the following table:

LOL_ADV	1
SET_PATH	2
OXC_DONE	3
SET_PATH_DONE	4
SET_PATH_ACK	5

Table 9: Value of the message context

The methods are the constructor, the set and get methods to access each field, a *toString()* method enabling a clean printing of the header fields. The *printType()* method enables a user to get a string representation of the type instead of its byte value.

3.6.4.1.1 ANALYSIS OF THE CLASS DYNRECOVERY

DynRecovery is the class implementing the whole behavior of the protocol.

3.6.4.1.1.1 THE CORE METHODS

The method *init()* is used to register the protocol DynRecovery at the ONIC for notification, in case a failure occurs.

The method *callback()* is called by the ONIC when a link, fiber, or lambda failure is detected. With this information, the protocol DynRecovery is able to determine the route, the channel, and the segment of the failure. Then it builds a *DynRecoveryHeader* to handle the Loss Of Light with a call to the HandleLOL_ADV method (see Table 10), using the DynRecoveryHeader to pass it the useful information.

All messages are processed by DynRecovery message handling functions and are send back to the ONIC by the method *transmit()*. This method is used to determine the next hop where the *DynRecoveryHeader* has to be sending to. Then it is pushed it down to the ONIC (via the protocol stack) to finally transmit the message.

When receiving a message in a node, a *DynRecoveryHeader* message is received by the protocol by using the method **push(message)**.

The push method drops the optical frame header, and filters the messages depending on their type, then delivers them to the right message handling method.

3.6.4.1.1.2 THE MESSAGE HANDLING METHODS.

There are five different types of messages and their message handling methods as shown in Table 10.

Message type	Handling method
LOL_ADV	handleLOL_ADV
SET_PATH	handleSET_PATH
OXC_DONE	handleOXC_DONE
SET_PATH_DONE	handleSET_PATH_DONE
SET_PATH_ACK	handleSET_PATH_ACK

Table 10: The Message Types

All of them are divided in two parts, one for a usual node, and one part for either the source or the destination node, depending on the messages propagation-direction. Their behavior is consistent with the functional description and is well commented in the coding.

3.6.4.1.1.3 THE TOOLS

The class **HandledRoutesHTable** is used at the node where the failure is detected to avoid that several calls to the call back method trigger several recovery mechanisms when only one is required.

The class **OXCHashTable** is used at the source node of the route in order to identify, which OXCs have already answered the SET_PATH message with the answer message SET_OXC_DONE.

The method *printDynRecoveryMessage* is used to create a preformatted printout of the DynRecovery messages.

The method *printRoute (OpticalRoute)* is used to print out all the nodes of the first path that the given route is using.

3.6.4.1.1.4 THE TOOLS USED TO COMPUTE THE NEXT HOP

The method *transmit* only accepts a DynRecoveryHeader as argument; it needs this header to compute the forwarding information.

The only information it needs is the id of the next add-lambda. This information is provided by the method *getNextAddLambdaID.*

The method *getNextAddLambdaID* browses through the list of all the add-lambdas of each ONIC in the node until the node at the opposite side of the lambda connected to this add-lambda matches the next hop for the dynamic recovery "DynRecovery" (the match is done with the node IDs)

The node at the opposite side of the lambda connected to this add-lambda is found by using the method *getADLDestNodeID*.

The next node for DynRecovery is found with the method *getNextHostID*.

The method *getNextHostID* uses the method *getTransmissionFiber* because with a given DynRecoveryHeader it is more convenient to find the next transmission fiber.

3.6.4.1.1.5 OTHER TOOLS USED BY THE MESSAGE HANDLING METHODS

The methods *plugAtDestination* and *plugAtSource* are used to register the backup route at the source and at the destination node. The precondition is that the number of lambdas and their bandwidth is the same as it is in the original route.

The method *isDownStreamOfFailure* is more specifically used by the method *callback* to determine, if the current node is located upstream or downstream of the failure. If the node is located upstream, no action is undertaken, whereas if the node is located downstream the normal action is processed.

The method *getTargetNode* is used to get the direction of the traffic flow.

3.6.4.2 DYNRECOVERY ADVANCED FEATURES.

3.6.4.2.1 HANDLING OF SEVERAL ROUTE BACKUP PROCESSES

It is possible that at the same time in each node are multiple backup processes are active, even if only one instance of the DynRecovery protocol is installed.

The current implementation of *DynRecovery* handles this problem by maintaining a *HashTable* that contains the already handled routes. This *Hashtable* is only used in the source node to know for which failed route all the OXCs are answering to. Another similar *HashTable* is used in the node where the failure is detected. Indeed there is only one failure recovery process per route. Thus, if this node gets another failure notification from an upstream link of the same route it doesn't start any new action.

The failure will be detected by the *ONIC* and delivered to the instance of *DynRecovery* by passing the failed or recovered lambda or fiber.

If the optical fiber contains several wavelengths, used by the same route, DynRecovery is going to receive as many callbacks for this route as there are used wavelengths. As explained above the protocol only starts one recovery process and recovers the whole route, not only the failed lambda channel.

At any moment, a node can handle a number of routes theoretically infinite. Indeed the forwarding mechanism is packet based (every packet contains all the necessary information to reach its destination). A basic piece of information that every *DynRecovery* packet is carrying is the route number along which it is traveling.

3.6.4.2.2 BI-DIRECTIONAL AND UNIDIRECTIONAL ROUTES

DynRecovery is able to handle either bidirectional or unidirectional routes. The signaling channel used is shared neither with the route nor with the backup route. The signaling channel requires being bidirectional whereas the route can be unidirectional.

3.6.4.3 COMMENTS

3.6.4.3.1 FUTURE IMPROVEMENTS

The current implementation of the dynamic recovery protocol might require a few improvements in the future: if it is not kept at the lowest level above the optical layer nothing prove that the signaling channels will behave as a fair queuing systems. This can be a problem if an OXC_DONE message arrives before the SET_PATH message at the source node.

The methods *plugAtSource* and *plugAtDestination* should be improved to work with any kind of lambda used.

After the backup route is installed, the switches of the original working path are not resetted. As a result after a recovery of a failure the resources are not freed.

3.6.4.3.2 POSSIBLE EXTENSIONS

The current implementation of *DynRecovery* can be easily extended in many ways to build route recovery protocols.

For instance, for tests reasons, DynRecovery has been brought above UDP/IP (it is a little fake because the UDP and IP protocols are not used, just the headers are used to dump packets in the ONICS). Still the class **DynRecoveryUDPIP** is an example of how *DynRecovery* can be overwritten and extended.

3.6.5 DML SCHEMAS

Table 11 contains the DML schema for the protocol followed by some comments about the parameters.

<pre>ProtocolSession [
  name DynRecovery
  use
gov.nist.antd.merlin.protocol.signaling.DynRecovery
  BCKUPALGO ShortestPathDistanceSRLG
  debug $S
  message $S
]</pre>
	Optional attribute BCKUPALGO specifies the name of the algorithm to use for backup.
	Optional attribute debug is used to print additional information.
	Optional attribute message specifies if the content of the control messages must be printed.

Table 11: Configuration Schema of the class DynRecovery

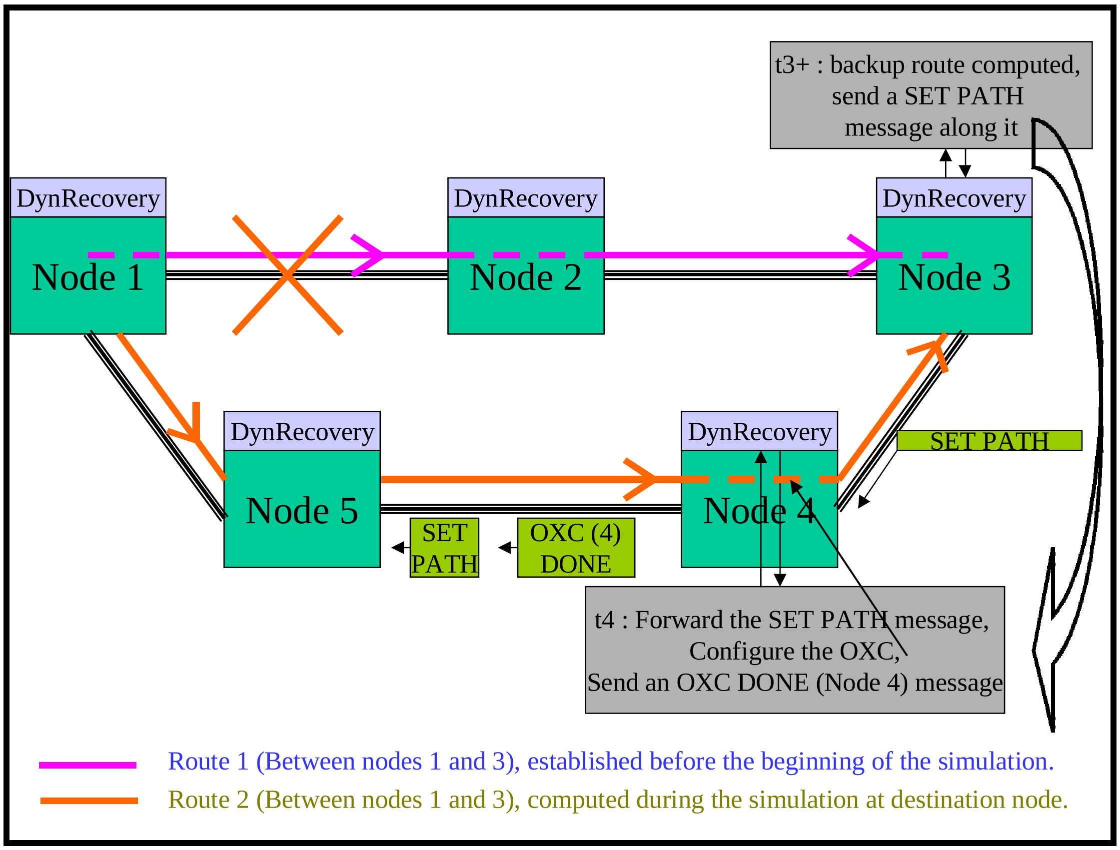
The attributes **debug**, **message** and **BCKUPALGO** are all optional. Their default values are respectively: *false*, false and *ShortestPathDistanceSRLG*.

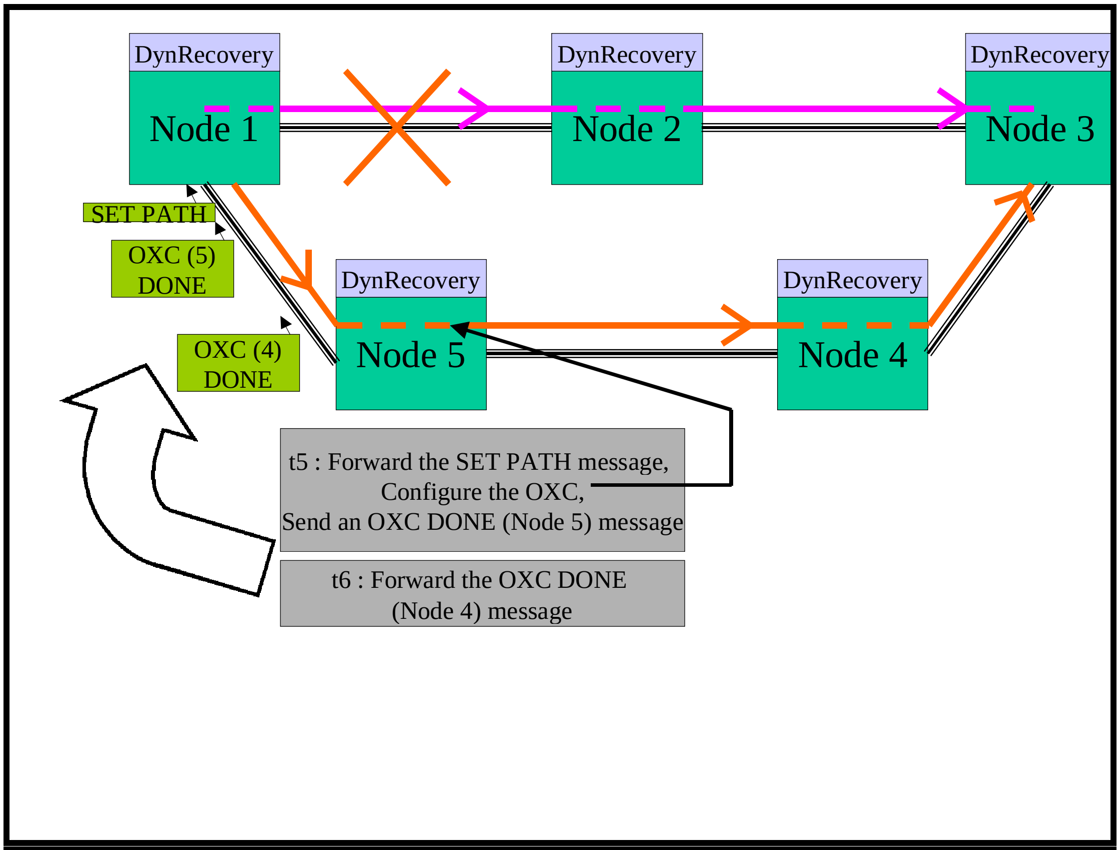
If the attribute *debug* is set to *true*, it means that all the debug information are going to be visible. The attribute “message == true” means this instance of DynRecovery is going to print the content of all the messages it handles. The attribute *BCKUPALGO* has to be declared in only one of all the nodes using the protocol *DynRecovery*. This attribute is network wide global and specifies which algorithm has to be used to generate backup routes. Here the default value is *ShortestPathDistanceSRLG*

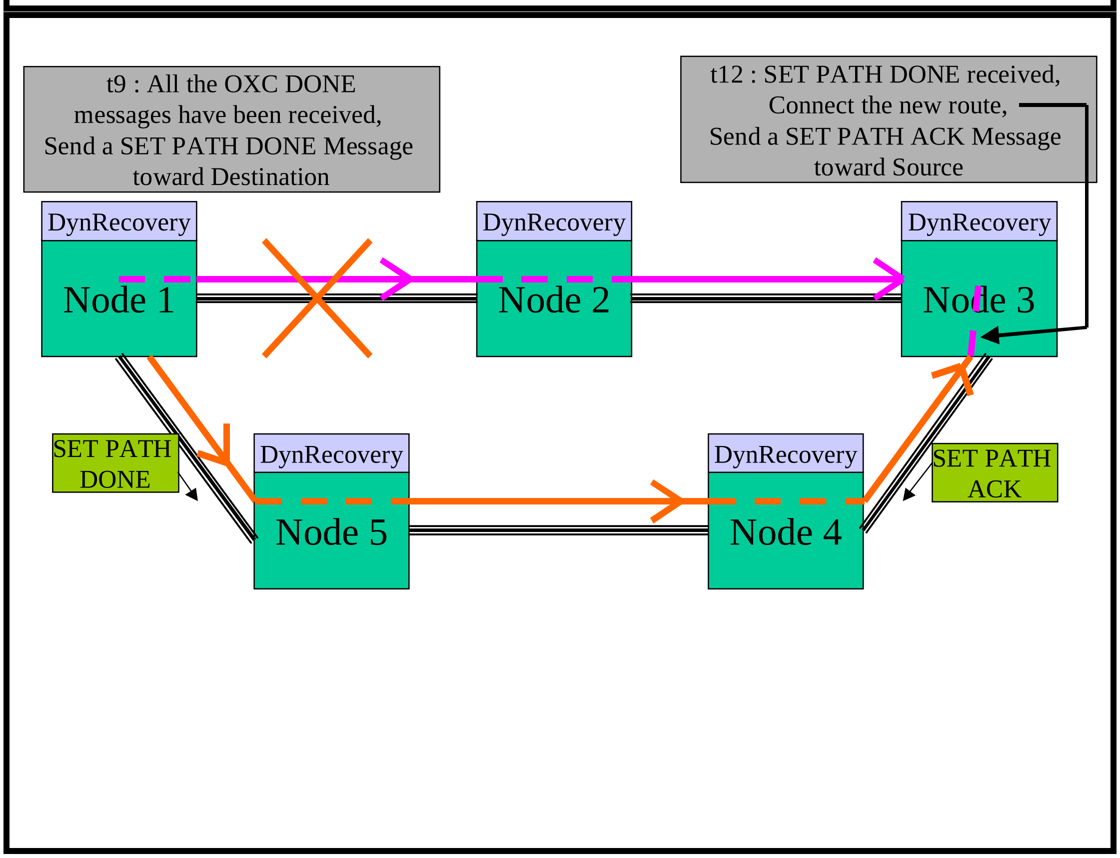
3.6.6 Example

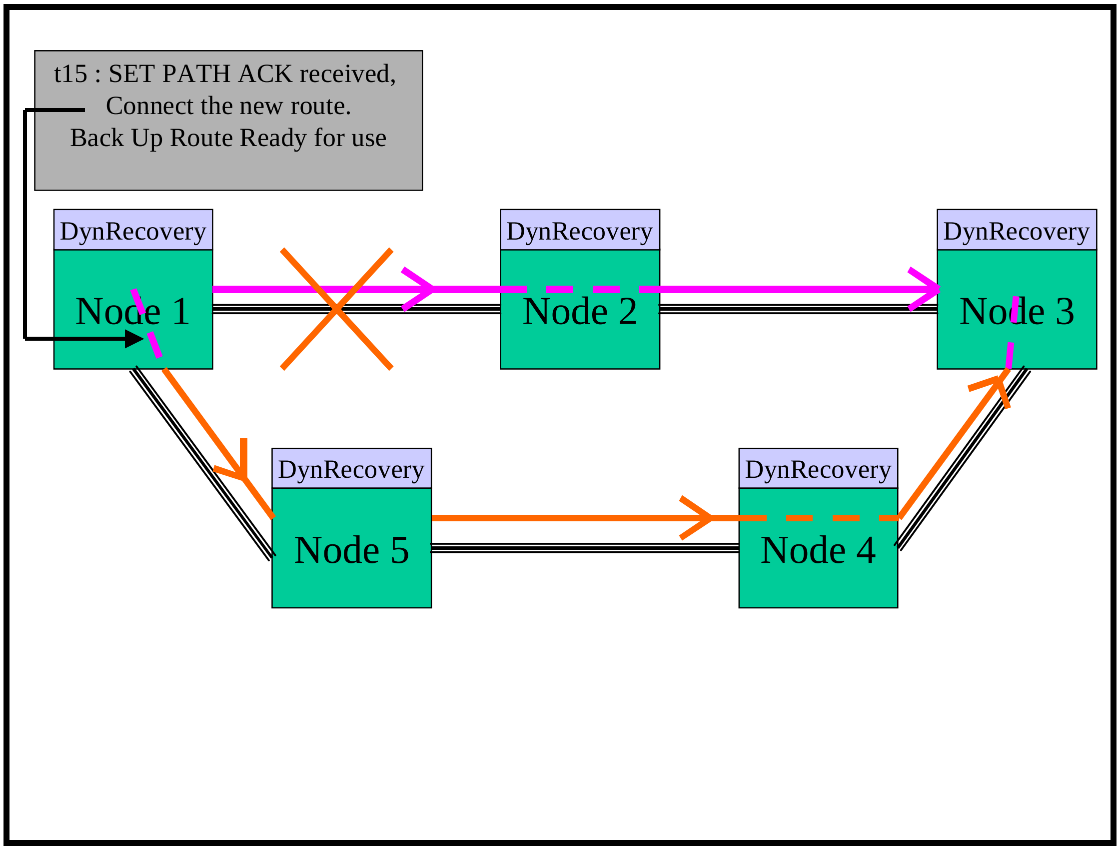
3.6.6.1 Typical behavior on a 5 node ring topology

3.6.6.1.1 Charts:




DynRecovery
Node 1
DynRecovery
Node 2
DynRecovery
Node 3
SET PATH
OXC (5)
DONE
DynRecovery
Node 5
DynRecovery
Node 4
OXC (4)
DONE
t5 : Forward the SET PATH message,
Configure the OXC,
Send an OXC DONE (Node 5) message
t6 : Forward the OXC DONE
(Node 4) message


t9 : All the OXC DONE
messages have been received,
Send a SET PATH DONE Message
toward Destination
t12 : SET PATH DONE received,
Connect the new route,
Send a SET PATH ACK Message
toward Source
DynRecovery
Node 1
DynRecovery
Node 2
DynRecovery
Node 3
SET PATH
DONE
DynRecovery
Node 5
DynRecovery
Node 4
SET PATH
ACK



3.6.6.1.2 TIMELINE OF THE 5-NODE RING TOPOLOGY

- **t1**: A failure occurs on the link between node 1 and 2,
- **t2**: On both sides of this link, the connected *ONIC*s signal the failure to the instance of *DynRecovery* of their node. The *DynRecovery* detects if its node is upstream or downstream of the failure.

  If it is downstream of the failure it sends a failure notification (**LOL_ADV**: Loss Of Light Advertisement) to the destination node of the route on which the failure has occurred, otherwise no action is undertaken.
- **t3**: The **LOL_ADV** message reaches the destination node. *DynRecovery* computes a backup route.

  The user can specify the backup algorithm used (see Table 11). The default algorithm is the ShortestPathDistanceSRLG. The Shared Risk Link Groups (SRLG) of the links are used to prevent the backup route from using the same links as the original route. If the

SRLGs of the links are missing in the configuration file (DML file), it occurs that the same link can be used if still available and usable resources are available. Depending on the algorithm used for the restoration, the results will be different.

Once the backup route is computed, *DynRecovery* sends a **SET_PATH** message along this route.

- **t4**: The **SET_PATH** message is received at node 4. It is directly forwarded toward the source node and then the OXC 4 configures itself (it effectively connects the lambdas). Once this configuration is done, *DynRecovery* sends an **OXC_DONE** message towards the source node.
- **t5**: The **SET_PATH** message is received and forwarded at node 5. The OXC 5 is configured, and an **OXC_DONE** message is sent towards the source node.
- **t6**: The **OXC_DONE** message from the node 4 is received in the node 5 and directly forwarded to the source node.
- **t7**: The source node receives the **SET_PATH** message and then gets ready to receive the **OXC_DONE** messages for each OXC of the backup route.
- **t8**: The source node receives the **OXC_DONE** message sent by the OXC 5.
- **t9**: The source node receives the **OXC_DONE** message sent by the OXC 4. When all the OXCs of the backup route have answered, the same node sends back a set **SET_PATH_DONE** message towards the destination node.
- **t10**: The node 5 receives the **SET_PATH_DONE** message and directly transmits it towards the destination node.
- **t11**: The node 4 receives the **SET_PATH_DONE** message and directly transmits it towards the destination node.
- **t12**: The destination node receives the **SET_PATH_DONE** message. It connects the backup route to the protocol previously using the original route. Then it sends backs a **SET_PATH_ACK** message towards the source node.
- **t13**: Node 4 receives the **SET_PATH_ACK** message and forwards it towards the source node.
- **t14**: Node 5 receives the **SET_PATH_ACK** message and forwards it towards the source node.

- **t15**: The source node receives the **SET_PATH_ACK** message. It connects the backup route to the protocol previously using the original route.

4 Acronyms

GLASS	GMPLS Lightwave Agile Switching Simulator
SSF	Scalable Simulation Framework
SSFNet	Scalable Simulation Framework Network
GMPLS	Generalized Multi-Protocol Label Switching
IP	Internet Protocol
DML	Data Modeling Language
OXC	Optical Cross Connect
ONIC	Optical Network interface Card

5 References

[1] Protocol modeling with SSF.OS

By Renesys Corporation

URL: http://www.ssfnet.com/InternetDocs/ssfnetTutorialProtocols.html

[2] How to implement a ProtocolSession in SSFNet

Borchert-Rouil, NIST/ANTD.

URL: http://www.antd.nist.gov/glass/

[3] OXCSwitch configuration in GLASS Simulator

Borchert-Rouil, NIST/ANTD.

URL: http://www.antd.nist.gov/glass/